**CS230 Exam 3**

**Author:** Diana Eastman

**Date Submitted:** April 25, 2013

**Notes:**

1. No known bugs.

2. The programs work as indicated in the comments.

3. To demonstrate testing, a soft copy of the testing script is included.

4. Exam 3 methods start on page 10

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* AdjMatGraphPlus.java

\* Created by: Diana Eastman

\* Created: April 25, 2013

\* Purpose: Implementation of the GraphPlus.java interface using

\* adjacency matrix of booleans for CS 230 Exam 3. Executes

\* non-recursive DFS and BFS, as well as two topological sorting

\* algorithms. Checks if a graph contains cycles using a modified

\* version of DFS. Builds off of HW 7 solutions provided by

\* Professor Metaxas.

\*

\* KNOWN FEATURES/BUGS:

\* It handles unweighted graphs only, but it can be extended

\* It does not handle operations involving non-existing vertices

\*

\* Correctness of program: I believe my program works correctly.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import javafoundations.\*;

import java.util.\*;

import java.io.\*;

public class AdjMatGraphPlus<T> implements GraphPlus<T>

{

private final int NOT\_FOUND = -1;

private final int DEFAULT\_CAPACITY = 1; // Small so that we can test expand

private int n; // number of vertices in the graph

private boolean[][] arcs; // adjacency matrix of arcs

private T[] vertices; // values of vertices

private int[] marks; // values of marked vertices used in modified DFS

private GraphPlus<T> cloneFromTGF; // clone of current graph

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Constructor. Creates an empty graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public AdjMatGraphPlus()

{

n = 0;

this.arcs = new boolean[DEFAULT\_CAPACITY][DEFAULT\_CAPACITY];

this.vertices = (T[])(new Object[DEFAULT\_CAPACITY]);

marks = new int[DEFAULT\_CAPACITY];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Second constructor:

\* Creates a new graph using the data found in a .tgf file.

If the file does not exist, a message is printed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public AdjMatGraphPlus(String tgf\_file\_name) {

//reset current graph

vertices = (T[]) (new Object[DEFAULT\_CAPACITY]);

arcs = new boolean[DEFAULT\_CAPACITY][DEFAULT\_CAPACITY];

marks = new int[DEFAULT\_CAPACITY];

n = 0;

try{

Scanner fileReader = new Scanner(new File(tgf\_file\_name));

while (!fileReader.next().equals("#")){

T line = (T) fileReader.next();

addVertex(line);

}

while (fileReader.hasNext()){

int arcVertex1 = fileReader.nextInt();

int arcVertex2 = fileReader.nextInt();

addArc(vertices[arcVertex1 -1], vertices[arcVertex2 -1]);

}

} catch (IOException ex) {

System.out.println(" \*\*\*(T)ERROR\*\*\* The file was not found: " + ex);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns true if the graph is empty and false otherwise.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isEmpty()

{

return (n == 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns the number of vertices in the graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public int n()

{

return n;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns the number of arcs in the graph by counting them.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public int m()

{

int total = 0;

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

if (arcs[i][j]) total++;

return total;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns true iff a directed edge exists from v1 to v2.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isArc (T vertex1, T vertex2)

{ return arcs[getIndex(vertex1)][getIndex(vertex2)]; }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Helper. Returns true iff an arc exists between two given indices

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private boolean isArc (int index1, int index2)

{

if (indexIsValid(index1) && indexIsValid(index2))

return arcs[index1][index2] == true;

else return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns true iff an edge exists between two given vertices

which means that two corresponding arcs exist in the graph

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isEdge (T vertex1, T vertex2){

return (isArc(vertex1, vertex2) && isArc(vertex2, vertex1)); }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns true IFF the graph is undirected, that is, for every

pair of nodes i,j for which there is an arc, the opposite arc

is also present in the graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isUndirected(){

for (int i = 0; i < n(); i++)

for (int j = 0; j < n(); j++)

if (arcs[i][j])

if (!arcs[j][i])

return false;

return true;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Adds a vertex to the graph, expanding the capacity of the graph

if necessary. If the vertex already exists, it does not add it.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void addVertex (T vertex)

{

if (getIndex(vertex) == NOT\_FOUND) {

if (n == vertices.length)

expandCapacity();

vertices[n] = vertex;

for (int i = 0; i <= n; i++)

{

arcs[n][i] = false;

arcs[i][n] = false;

}

n++;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Helper. Creates new arrays to store the contents of the graph

with twice the capacity.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private void expandCapacity()

{

T[] largerVertices = (T[])(new Object[vertices.length\*2]);

boolean[][] largerAdjMatrix =

new boolean[vertices.length\*2][vertices.length\*2];

int[] largerMarks = new int[marks.length\*2]; // Add in code to expand the array of marks as well

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

largerAdjMatrix[i][j] = arcs[i][j];

}

largerVertices[i] = vertices[i];

largerMarks[i] = marks[i];

}

vertices = largerVertices;

arcs = largerAdjMatrix;

marks = largerMarks;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Removes a single vertex with the given value from the graph.

Uses equals() for testing equality

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void removeVertex (T vertex)

{

for (int i = 0; i < n; i++)

if (vertex.equals(vertices[i]))

removeVertex(i);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Helper. Removes a vertex at the given index from the graph.

Note that this may affect the index values of other vertices.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private void removeVertex (int index)

{

if (indexIsValid(index))

{

n--;

for (int i = index; i < n; i++)

vertices[i] = vertices[i+1];

for (int i = index; i < n; i++)

for (int j = 0; j <= n; j++)

arcs[i][j] = arcs[i+1][j];

for (int i = index; i < n; i++)

for (int j = 0; j < n; j++)

arcs[j][i] = arcs[j][i+1];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Inserts an edge between two vertices of the graph.

If one or both vertices do not exist, ignores the addition.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void addEdge (T vertex1, T vertex2)

{

// getIndex will return NOT\_FOUND if a vertex does not exist,

// and the addArc calls will not insert it

addArc (getIndex(vertex1), getIndex(vertex2));

addArc (getIndex(vertex2), getIndex(vertex1));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Inserts an arc from vertex1 to vertex2.

If the vertices exist, else does not change the graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void addArc (T vertex1, T vertex2){

addArc (getIndex(vertex1), getIndex(vertex2));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Helper. Inserts an edge between two vertices of the graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private void addArc (int index1, int index2)

{

if (indexIsValid(index1) && indexIsValid(index2))

arcs[index1][index2] = true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Removes an edge between two vertices of the graph.

If one or both vertices do not exist, ignores the removal.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void removeEdge (T vertex1, T vertex2)

{

removeArc (getIndex(vertex1), getIndex(vertex2));

removeArc (getIndex(vertex2), getIndex(vertex1));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Removes an arc from vertex v1 to vertex v2,

if the vertices exist, else does not change the graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void removeArc (T vertex1, T vertex2){

removeArc (getIndex(vertex1), getIndex(vertex2)); }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Helper. Removes an arc from index v1 to index v2.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private void removeArc (int index1, int index2)

{

if (indexIsValid(index1) && indexIsValid(index2))

arcs[index1][index2] = false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns the index value of the first occurrence of the vertex.

Returns NOT\_FOUND if the key is not found.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private int getIndex(T vertex)

{

for (int i = 0; i < n; i++)

if (vertices[i].equals(vertex))

return i;

return NOT\_FOUND;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns the vertex object that is at a certain index

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private T getVertex(int v)

{

return vertices[v];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns true if the given index is valid.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private boolean indexIsValid(int index)

{

return ((index < n) && (index >= 0));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Retrieve from a graph the vertices x pointing to vertex v (x->v)

and returns them onto a linked list

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> getPredecessors(T vertex){

LinkedList<T> neighbors = new LinkedList<T>();

int v = getIndex(vertex);

for (int i = 0; i < n; i++)

{

if (arcs[i][v])

neighbors.add(getVertex(i)); // if T then add i to linked list

}

return neighbors;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Retrieve from a graph the vertices x following vertex v (v->x)

and returns them onto a linked list

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> getSuccessors(T vertex){

LinkedList<T> neighbors = new LinkedList<T>();

int v = getIndex(vertex);

for (int i = 0; i < n; i++)

{

if (arcs[v][i])

neighbors.add(getVertex(i)); // if T then add i to linked list

}

return neighbors;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Returns a string representation of the graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public String toString()

{

if (n == 0)

return "Graph is empty";

String result = new String("");

result += "Arcs\n";

result += "-----\n";

result += "i ";

for (int i = 0; i < n; i++)

{

result += "" + getVertex(i);

if (i < 10)

result += " ";

}

result += "\n";

for (int i = 0; i < n; i++)

{

result += "" + getVertex(i) + " ";

for (int j = 0; j < n; j++)

{

if (arcs[i][j])

result += "1 ";

else

result += "- "; //just empty space

}

result += "\n";

}

return result;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Saves the current graph into a .tgf file.

\* If it cannot save the file, a message is printed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void saveTGF(String tgf\_file\_name) {

try {

PrintWriter writer = new PrintWriter(new File(tgf\_file\_name));

//prints vertices by iterating through array "vertices"

for (int i = 0; i < n(); i++){

if (vertices[i] == null){

break;

}

else{

writer.print((i+1) + " " + vertices[i]);

writer.println("");

}

}

writer.print("#"); // Prepare to print the edges

writer.println("");

//prints arcs by iterating through 2D array

for (int i = 0; i < n(); i++){

for (int j = 0; j < n(); j++){

if (arcs[i][j] == true){

writer.print((i+1) + " " + (j+1));

writer.println("");

}

}

}

writer.close();

} catch (IOException ex) {

System.out.println("\*\*\*(T)ERROR\*\*\* The file could nt be written: " + ex);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* START OF EXAM 3 CODE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Clones the graph by saving the current one on the disk

\* as TEMP.tgf using saveTGF() and creating a new one using the

\* second constructor.

\* Should be called prior to running topological sort because it

\* TS is a destructive method and results in an empty graph.

\* Topological sort should be performed on the cloned graph.

\* @return the new graph.

\* FEATURE: It does not try to delete the file from the disk

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public GraphPlus<T> clone()

{

saveTGF("TEMP.tgf");

cloneFromTGF = new AdjMatGraphPlus<T>("TEMP.tgf");

return cloneFromTGF;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Getter method to call the clone function and return the cloned

\* graph.

\* @returns the cloned graph.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public GraphPlus<T> getClone()

{

clone();

return cloneFromTGF;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Checks if a vertex is a sink, (points to no other vertex)

\* A vertex is a sink if it has no successors, so the getSuccessors

\* method is called and the returning LinkedList is checked.

\* If the LinkedList is empty (size == 0), the vertex is a sink.

\* PREREQUISITE: The vertex exists in the graph.

\* @return true if the vertex is a sink, false if it is not.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isSink(T vertex)

{

LinkedList<T> sinks = new LinkedList<T>();

if (getIndex(vertex)!= NOT\_FOUND){

sinks = getSuccessors(vertex);

return (sinks.size() == 0);

}

else

System.out.println(vertex + " does not exist in this graph.");

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Retrieves the vertices that are sinks by applying the method

\* isSink() to every vertex in the vertices array.

\* If the LinkList of successors is empty (isSink() returns

\* true) the vertex is a sink and is added to the sinks LinkedList

\* @return all the sinks in a linked list

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> allSinks()

{

LinkedList<T> sinks = new LinkedList<T>();

for (int i=0; i< n; i++){

if(isSink(vertices[i]))

sinks.add(vertices[i]);

}

return sinks;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Checks if a vertex is a source, (no vertex points to it)

\* A vertex is a source if it has no predecessors, so the

\* getPredecessors() method is called and the returning LinkedList

\* is checked. If the LinkedList is empty (size == 0), the vertex

\* is a source.

\* PREREQUISITE: The vertex exists in the graph.

\* @return true if the vertex is a source, false if it is not

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isSource (T vertex)

{

LinkedList<T> sources = new LinkedList<T>();

if (getIndex(vertex)!= NOT\_FOUND){

sources = getPredecessors(vertex);

return (sources.size() == 0);

}

else

System.out.println(vertex + " does not exist in this graph.");

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Retrieves the vertices that are sources by applying the method

\* isSource() to every vertex in the vertices array.

\* If the LinkList of predecessors is empty (isSource() returns

\* true) the vertex is a source and is added to the sources LinkedList

\* @return all the sources in a linked list

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> allSources()

{

LinkedList<T> sources = new LinkedList<T>();

for (int i=0; i< n; i++){

if(isSource(vertices[i]))

sources.add(vertices[i]);

}

return sources;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Checks if a vertex is a isolated

\* A vertex is isolated if it is both a sink AND a source, so if

\* both isSink() and isSource() are true, the vertex is isolated

\* @return true if the vertex is isolated, false if it is not

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean isIsolated (T vertex)

{

return (isSink(vertex) && isSource(vertex));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Topologically sorts the vertices of a DAG by selecting a

\* source vertex v, removing v and all edges leading from it,

\* adding v to the end of a linked list of vertices, and repeating

\* until there are no vertices left.

\* Should be called on the CLONED graph.

\*

\* PREREQUISITE: The input graph must be a DAG, i.e., NO CYCLES.

\* @return the topologically sorted vertices in a linked list and

\* an empty linked list if the input graph is not a DAG

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> topologicalSort()

{

LinkedList<T> sorted = new LinkedList<T>();

if (containsCycle()) { // Empty linked list returned if graph contains cycles - no sort executed

System.out.println("Graph contains cycles. Topological sort will not be performed.");

return sorted;

}

while(n>0) {

LinkedList<T> sources = new LinkedList<T>(allSources());

for (int i=0; i < sources.size(); i++){

T vertex = sources.get(i);

sorted.add(vertex); // Add to the end of the LL

removeVertex(vertex); // Calling removeVertex() will vertex and its edges from the graph

}

}

return sorted;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Topologically sorts the vertices of a DAG by selecting a

\* sink vertex v, removing v and all edges pointing to it,

\* adding v to the front of a linked list of vertices, and repeating

\* until there are no vertices left.

\* Should be called on the CLONED graph.

\*

\* PREREQUISITE: The input graph must be a DAG, i.e., NO CYCLES.

\* @return the topologically sorted vertices in a linked list and

\* an empty linked list if the input graph is not a DAG

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> topologicalSortAlt()

{

LinkedList<T> sorted = new LinkedList<T>();

if (containsCycle()) { // Empty linked list returned if graph contains cycles - no sort executed

System.out.println("Graph contains cycles. Topological sort will not be performed.");

return sorted;

}

while(n>0) {

LinkedList<T> sinks = new LinkedList<T>(allSinks());

for (int i=0; i < sinks.size(); i++){

T vertex = sinks.get(i);

sorted.addFirst(vertex);

removeVertex(vertex);

}

}

return sorted;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Returns a LinkedList contining a DEPTH first search traversal

\* starting at the given index. Where vertex v has multiple

\* successors, the vertex that appears in the vertices array

\* first will be visited first. Implementation is Stack-based and

\* non-recursive.

\* If the index is not valid, it returns an empty list

\* @return a linked list with the vertices in depth-first order

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> DFS(T vertex)

{

T currentvertex;

LinkedStack<T> traversalStack = new LinkedStack<T>();

LinkedList<T> resultList = new LinkedList<T>();

boolean [] visited = new boolean[n]; // Indices of visited elements match the indices of the vertices array

boolean found;

if (getIndex(vertex)== NOT\_FOUND) // If index not found, return empty linked list

return resultList;

for (int i=0; i<n; i++)

visited[i] = false; // Initialize all boolean array visited to false

traversalStack.push(vertex); // Put starting vertex on the stack and in the resultList; mark as visited

resultList.add(vertex);

visited[getIndex(vertex)] = true;

while (!traversalStack.isEmpty()) // While there are still items in the stack, peek at vertex

{

currentvertex = traversalStack.peek();

found = false;

for (int i = 0; i < n && !found; i++) // Finds an arc leading from the vertex that has not been visited

if(isArc(currentvertex, vertices[i]) && !visited[i])

{

traversalStack.push(vertices[i]);

resultList.add(vertices[i]);

visited[i] = true; // Vertex marked as visited after it has been added to the result list

found = true; // Found = true, exits the loop; last vertex added is new top of the stack

}

if (!traversalStack.isEmpty() && !found) // If a dead end is reached, pop an element of the stack and back track

traversalStack.pop();

}

return resultList;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Returns a LinkedList containing BREADTH first search traversal

\* starting at the given index. Where vertex v has multiple

\* successors, the vertex that appears in the vertices array

\* first will be visited first. Implementation is queue-based and

\* non-recursive.

\* If the index is not valid, it returns an empty list

\* @return a linked list with the vertices in breadth-first order

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public LinkedList<T> BFS(T vertex)

{

T currentvertex;

LinkedQueue<T> traversalQueue = new LinkedQueue<T>();

LinkedList<T> resultList = new LinkedList<T>();

boolean [] visited = new boolean[n]; // Indices of visited elements match the indices of the vertices array

if (getIndex(vertex)== NOT\_FOUND) // If index not found, return empty linked list

return resultList;

for (int i=0; i<n; i++)

visited[i] = false; // Initialize boolean array visited to false

traversalQueue.enqueue(vertex); // Add start index to the queue and mark it as visited

visited[getIndex(vertex)] = true;

while(!traversalQueue.isEmpty()){ // While there are still items in the queue, dequeue vertex, add to result list

currentvertex = traversalQueue.dequeue();

resultList.add(currentvertex);

for (int i = 0; i < n; i++) // Finds all of arcs leading from the vertex that have not been visited

if (isArc(getIndex(currentvertex), i) && !visited[i])

{

traversalQueue.enqueue(getVertex(i)); // Add to queue

visited[i] = true; // Mark as visited

}

}

return resultList;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Checks if a graph contains any cycles by performing

\* a modified DFS on the graph, starting with the first node

\* in the vertices array.

\* KEY:

\* 0 = the vertex has not been visited previously

\* 1 = in the process of examining the vertex's successors

\* 2 = a vertex whose successors have all been processed (has no

\* back edges)

\*

\* @return true if the graph contains any cycles

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean containsCycle()

{

for (int i=0; i<n; i++)

marks[i] = 0; // Store the status of the vertex as it is encountered - all are initialized as unvisited

for (int i=0; i<n; i++) {

if (marks[i]==0) {

if (visit(i)){ // If visit() returns true, the graph contains at least one cycle

return true;

}

}

}

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Checks if a vertex is seen twice before all of its successors

\* have been visited. Predicated on the fact that in DFS,

\* a node whose successors have all been visited can be seen more

\* than once--even if no cycle exists. If a vertex is seen twice

\* before its successors have all been visited (i.e, it is visited

\* again while marked as number 1, as per below), then the graph

\* contains at least 1 cycle. If all of the elements in the

\* marks array are 2, the graph is a DAG.

\*

\* @return true if the graph contains any cycles

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public boolean visit(int index){

marks[index] = 1; // Mark the index of the intial vertex as in process

for (int i=0; i<n; i++){

if(isArc(index,i)){ // isArc() returns true IFF there is a directed edge from index->i

if (marks[i]==1) // Encountered a vertex before its successors have all been processed, indicating a cycle

return true;

else if (marks[i]==0) { // Encountered a vertex that has not been visited, recursively call visit()

if(visit(i))

return true;

}

}

}

marks[index] = 2; // All the successors have been processed without encountering back edges

return false;

}

}